On the Power of Statistical Model Checking

Kim Guldstrand Larsen, Axel Legay

On the Power of Statistical Model Checking *

Kim G. Larsen[1,2] and Axel Legay[1,2]

[1] Aalborg University
[2] Inria

Abstract. This paper contains material for our tutorial presented at STRESS 2016. This includes an introduction to Statistical Model Checking algorithms and their rare event extensions, as well as an introduction to two well-known SMC tools: PLASMAand UPPAAL.

1 Context

This paper summarizes the content of our STRESS tutorial on Statistical Model Checking (SMC). More details about can be found in the papers presented in the SMC session at ISOLA 2016 (part of the same volume).

In short, in order to solve the model checking problem for probabilistic systems, the SMC approach *simulates* the system for finitely many runs, and use *hypothesis testing* or estimators to infer whether the samples provide *statistical* evidence for the satisfaction or violation of the specification [35, 44].

SMC is thus based on the notion that since sample runs of a stochastic system are drawn according to the distribution defined by the system, they can be used to obtain estimates of the probability measure on executions. Starting from time-bounded Probabilistic Bounded Temporal Logic properties (PCTL) [44], the technique has been extended to handle properties with unbounded until operators [39], as well as to black-box systems [38, 44]. Tools, based on this idea have been built [28, 40, 44], and have been used to analyse many systems that are intractable numerical approaches.

In this tutorial, we first introduce two SMC algorithms that work with a finite set of observations. The first one, which is based on hypothesis testing, can be used to check whether the probability to satisfy the property exceed some fixed valued. The second algorithm, which is based on Monte Carlo, can be used to estimate this probability. For the estimation case, it is well-known that Monte Carlo based approaches have trouble to estimate very low probabilities. To overcome this difficulty, we introduce two major extensions of Monte Carlo that are importance splitting and sampling. Those can be used to estimate very low probability either by modifying the probability mass of the system, or by guiding the executions with respect to the property to be verified.

* The research has received funding from the European FET projects SENSATION, Grant Agreement № 2888917 (DALI), and CASSTING, the Sino-Danish Basic Research Center IDEA4CPS , the Danish Innovation Center DiCyPS , as well as the ERC Advanced Grant LASSO and a CREATIVE Grant from the Brittany region.

In the second part of the tutorial, we will introduce tools and SMC applications. The first tool that will be introduced is Plasma. This tool is an efficient self-contained SMC tool and software library [12] written in Java. We will illustrate the portability of the tool with two applications coming from the assisted living and energy-centric worlds, respectively. Then, we will turn our focus to those systems whose behavior also depends on real-time information. We will introduce a SMC extension for the well-known tool Uppaal. We will also show how the tool can be extended to synthesize good strategies for those systems whose behavior depends on both timed and stochastic informations. The rest of this paper gives some pointers to the content of the tutorial.

2 Statistical Model Checking: Algorithms

2.1 Qualitative and quantitative original SMC algorithms

Consider a stochastic system $\mathcal{S}$ and a logical property φ that can be checked on finite executions of the system. Statistical Model Checking (SMC) refers to a series of simulation-based techniques that can be used to answer two questions: (1) *Qualitative*: Is the probability for $\mathcal{S}$ to satisfy φ greater or equal to a certain threshold? and (2) *Quantitative*: What is the probability for $\mathcal{S}$ to satisfy φ? In contrast to numerical approaches, the answer is given up to some correctness precision.

As we said above, SMC first consists in monitoring φ on a finite set of executions of the system. Then, an algorithm from the statistics is used to answer either the qualitative or the quantitative question. As we shall see in the rest of the section, the algorithm that is applied clearly depends on the question one wants to solve. It is important to notice that any SMC algorithm must terminates after producing a finite amount of executions, each of them being of finite length. As a consequence the answer of any SMC algorithm is correct up to some confidence.

In the sequel, we use B_i as a Bernouili variable associated with the ith simulation of the system. The outcome for B_i, denoted b_i, is 1 if the simulation satisfies φ and 0 otherwise. We also use $p = Pr(\varphi)$ as the true probability for the system to satisfy φ.

Qualitative Answer. The main approaches [38,44] proposed to answer the qualitative question are based on *sequential hypothesis testing* [41]. The idea is to reduce the qualitative question to the one of a test between two hypothesis. Concretely, to determine whether $p \geq \theta$, the algorithm will test the hypothesis $H : p \geq \theta$ against $K : p < \theta$.

As we shall see, the principle of any sequential hypothesis testing will be to simulate System $\mathcal{S}$ execution by execution. After each new execution, a check will be performed to decide between the two hypothesis. The algorithm will have to continue until a decision is taken (hence the term "sequential").

Of course, this decision must be taken after a finite number of executions have been monitored. Consequently, the algorithm may take the wrong decision.

One thus has to elaborate on the quality of the answer that is provided. In hypothesis testing, this is called the *strength* of the test. It is determined by two parameters, α and β, such that the probability of accepting K (respectively, H) when H (respectively, K) holds, called a Type-I error (respectively, a Type-II error) is less or equal to α (respectively, β). A test has *ideal performance* if the probability of the Type-I error (respectively, Type-II error) is exactly α (respectively, β).

Another difficulty with sequential testing algorithm if p is really close to θ, then it will take a lot of simulation to decide between the two hypothesis. In fact, one can even show that if the two quantities are infinitely close, then one cannot converge in finite amount of time. A solution to this problem is to use an *indifference region* $[p_1, p_0]$ (given some δ, $p_1 = \theta - \delta$ and $p_0 = \theta + \delta$) and to test $H_0 : p \geq p_0$ against $H_1 : p \leq p_1$. Intuitively, this means that if the two quantities are infinitely close, then it does not matter to select one or the other hypothesis.

We now sketch the Sequential Probability Ratio Test (SPRT) that is the most well-known hypothesis testing algorithm. In this algorithm, one fixes two values A and B. Let m be the number of observations that have been made so far. The test is based on the following quotient:

$$\frac{p_{1m}}{p_{0m}} = \prod_{i=1}^{m} \frac{Pr(B_i = b_i \mid p = p_1)}{Pr(B_i = b_i \mid p = p_0)} = \frac{p_1^{d_m}(1-p_1)^{m-d_m}}{p_0^{d_m}(1-p_0)^{m-d_m}},$$

where $d_m = \sum_{i=1}^{m} b_i$.

The idea is to accept H_0 if $\frac{p_{1m}}{p_{0m}} \geq A$, and H_1 if $\frac{p_{1m}}{p_{0m}} \leq B$. The algorithm computes $\frac{p_{1m}}{p_{0m}}$ for successive values of m until either H_0 or H_1 is satisfied. In the work of Wald, it is showed how to select A and B such that the strength of the test (see type-error above) is respected. In practice as A and B are correlated to α and β, the number of simulations to terminates highly depends on α and β. The smaller those two values are and the more simulations one will need to terminate.

Quantitative Answer. The objective of a quantitative answer is to estimate the probability p to satisfy the property. As we will only have a finite number of executions to monitor, the best one can obtain is an estimate $\hat{p}$ that is at some distance δ from the true probability. Of course, there is always the possibility that the algorithm does not respect this distance. One thus seek for a confidence α on our estimator[3].

To obtain such an estimator $\hat{p}$, we will use a so-called Monte Carlo estimator approach. The idea is quite simple. Let m be a pre-computed number of executions on which the property is to be monitored. We set $\hat{p}$ to be the number of executions that does satisfy the property divided by m. The challenge is now to select an m such that δ and α are guaranteed.

Given a *precision* δ, the *Chernoff bound* of [37] is used to compute a value for $\hat{p}$ such that $|\hat{p} - p| \leq \delta$ with *confidence* $1 - \alpha$. Let $\hat{p} = \sum_{i=1}^{m} b_i/m$, then the

[3] Please note that δ and α do not have the same meaning as for the qualitative question.

Chernoff bound [37] gives $Pr(|\hat{p} - p| \geq \delta) \leq 2e^{-2m\delta^2}$. As a consequence, if we take $m = \lceil \ln(2/\alpha)/(2\delta^2) \rceil$, then $Pr(|\hat{p} - p| \leq \delta) \geq 1 - \alpha$.

Observe that there is a major difference between the qualitative and quantitative algorithm, that is the quantitative algorithm pre compute the number of executions it needs to terminate. There are however, sequential versions of the quantitative algorithms. Their study goes beyond this paper.

2.2 Towards Rare Events: on Extending SMC Algorithms

Statistical model checking avoids the exponential growth of states associated with probabilistic model checking by estimating probabilities from multiple executions of a system and by giving results within confidence bounds. Rare properties are often important but pose a particular challenge for simulation-based approaches, hence a key objective for SMC is to reduce the number and length of simulations necessary to produce a result with a given level of confidence. In the literature, one finds two techniques to cope with rare events: *importance sampling* and *importance splitting*.

In order to minimize the number of simulations, importance sampling works by estimating a probability using weighted simulations that favour the rare property, then compensating for the weights. For importance sampling to be efficient, it is thus crucial to find good importance sampling distributions without considering the entire state space. In [29], we presented a simple algorithm that uses the notion of cross-entropy minimisation to find an optimal importance sampling distribution. In contrast to previous work, our algorithm uses a naturally defined low dimensional vector of parameters to specify this distribution and thus avoids the intractable explicit representation of a transition matrix. We show that our parametrisation leads to a unique optimum and can produce many orders of magnitude improvement in simulation efficiency.

One of the open challenges with importance sampling is that the variance of the estimator cannot be usefully bounded with only the knowledge gained from simulation. Importance *splitting* achieves this objective by estimating a sequence of conditional probabilities, whose product is the required result. In [30] we motivated the use of importance splitting for statistical model checking and were the first to link this standard variance reduction technique [32] with temporal logical. In particular, we showed how to create *score functions* based on logical properties, and thus define a set of *levels* that delimit the conditional probabilities. In [30] we also described the necessary and desirable properties of score functions and levels, and gave two importance splitting algorithms: one that uses fixed levels and one that discovers optimal levels adaptively.

One interesting aspect of rare events is that the performances of algorithms for single core often degenerate when moving to distributed architectures. Details on this study can be found in [31].

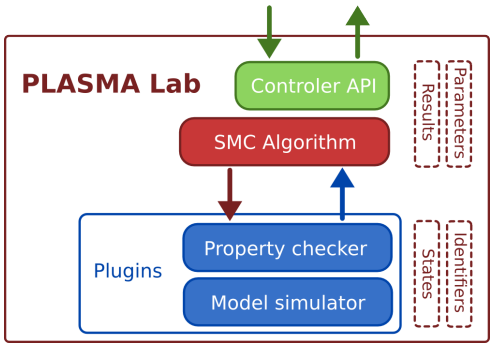


Fig. 1: Plasma architecture

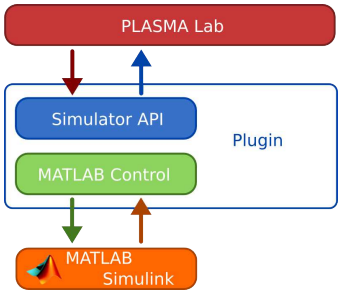


Fig. 2: Interface between Plasma and Simulink

3 Plasma: a Modular Toolset for Statistical Model Checking

Plasma is a compact, efficient and flexible platform for statistical model checking of stochastic models. The tool offers a series of SMC algorithms which includes classical SMC algorithms and rare events ones presented above. The main difference between Plasma and other SMC tools is that Plasma proposes an API abstraction of the concepts of stochastic model simulator, property checker (monitoring) and SMC algorithm. In other words, the tool has been designed to be capable of using external simulators, input languages, or SMC algorithms. This not only reduces the effort of integrating new algorithms, but also allows us to create direct plug-in interfaces with industry used specification tools. The latter being done without using extra compilers. Plasma is the focus of ongoing collaborations with companies Dassault, Thales, IBM, and EADS. Plasma is also used by several European projects.

Fig. 1 presents Plasma architecture. More specifically, the relations between model simulators, property checkers, and SMC algorithms components. The simulators features include starting a new trace and simulating a model step by step. The checkers decide a property on a trace by accessing to state values. They also control the simulations, with a *state on demand* approach that generates new states only if more states are needed to decide the property. A SMC algorithm component, such as the Monte Carlo algorithm, is a runnable object. It collect samples obtained from a checker component. Depending on the property language, their checker either returns Boolean or numerical values. The algorithm then notifies progress and sends its results through the Controller API.

Usage The GUI provides an integrated development environment (IDE) to facilitate the use of Plasma as a standalone statistical model checker with multiple 'drop-in' modelling languages. Plasma is usually invoked via its GUI. It may also be invoked from the command line or embedded in other software as a library. In addition to the GUI, Plasma provides an SMC engine in the form

of a pre-compiled jar file. A source template is also provided to create custom simulator classes. The minimum requirement to create a custom simulator is to implement methods that (*i*) initiate a new simulation and (*ii*) advance the simulation by one step. Dedicated language parsers are typically invoked in the constructor of the custom simulator class. In coordination with this architecture, we use a plugin system to load models and properties components. It is then possible to support new model or property languages. Adding a simulator, a checker or an algorithm component is pretty straigthforward as we will see with simulink below. One of the goal of PLASMA is also to benefit from a massive distribution of the simulations, which is one of the advantage of the SMC approach. Therefore PLASMA API provides generic methods to define distributed algorithms. We have used these functionalities to distribute large number of simulations over a computer grid [4].

3.1 Application to Motion Planning

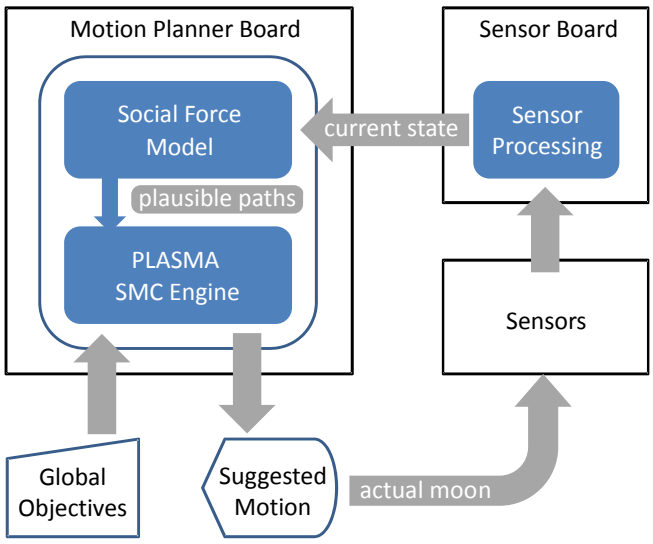


Fig. 3: Control loop of DALi motion planner.

PLASMA is used by the DALi project in a novel motion planning application of SMC. DALi aims to develop an autonomous device to help those with impaired ability to negotiate complex crowded environments (e.g. shopping malls). High level constraints and the objectives of the user are expressed in temporal logic, while low level behaviour is predicted by the 'social force model' [24]. The planner

[4] `https://project.inria.fr/plasma-lab/documentation/tutorial/igrida-experimentation/`

first hypothesises many plausible futures for a range of possible user actions, then chooses the action which maximises the probability of success.

Plasma was integrated with MATLAB to develop the prototype algorithm. The final version is implemented directly in C on embedded hardware and finds the optimum trajectory in a fraction of a second [15]. Plasma improves the social force model's ability to avoid collisions by a factor of five [15]. Using behavioural templates [14], the predictive power of our SMC-based motion planner can be even greater.

3.2 Integration Plasma with Simulink

We now show how to integrate Plasma within Simulink, hence lifting the power of our simulation approaches directly within the tool. We will focus on those Simulink models with stochastic information, as presented in [45]. But our approach is more flexible because the user will directly use Plasma within the Simulink interface, without third party.

Simulink is a block diagram environment for multi-domain simulation and Model-Based Design approach. It supports the design and simulation at the system level, automatic code generation, and the testing and verification of embedded systems. Simulink provides a graphical editor, a customizable set of block libraries and solvers for modeling and simulation of dynamic systems. It is integrated within MATLAB. The Simulink models we considered have special extensions to randomly behave like failures. By default the Simulink library provides some random generators that are not compatible with statistical model checking: they always generate the same random sequence of values at each execution. To overcome this limitation we use some C-function block calls that generate independent sequences of random draws.

Our objective was to integrate Plasma as a new Simulink library. For doing so, we developed a new simulator plugin whose architecture is showed in Fig. 2. One of the key points of our integration has been to exploit MATLAB Control[5], a library that allows to interact with MATLAB from Java. This library uses a proxy object connected to a MATLAB session. MATLAB invokes, *e.g.* functions `eval`, `feval` ... as well as variables access, that are transmitted and executed on the MATLAB session through the proxy. This allowed us to implement the features of a model component, controlling a Simulink simulation, in MATLAB language. Calls to this implementation are then done in Java from the Plasma plugin.

Regarding the monitoring of properties, we exploit the simulation output of Simulink. More precisely, BLTL properties are checked over the executions of a SDES, *i.e.*, sequences of states and time stamps based on the set of state variables SV. This set must be defined by declaring in Simulink signals as log output. During the simulation these signals are logged in a data structure containing time stamps and are then retrieved as states in Plasma. One important point is that Simulink discretizes the signals trace, its sample frequency being parameterized

[5] `https://code.google.com/p/matlabcontrol/`

by each block. In terms of monitoring this means that the sample frequency must be configured to observe any relevant change in the model. In practice, the frequency can be set as a constant value, or, if the model mixes both continuous data flow and state flow, the frequency can be aligned on the transitions, *i.e.*, when a state is newly visited.

Illustration of the integration Let us now illustrate the approach with a concrete example (see also `https://project.inria.fr/plasma-lab/`). This model is taken from the Simulink/Stateflow examples library. It describes the fuel control system of a gasoline engine. The system is made robust by detecting failures in sensors and dynamically re-configuring its behavior to maintain a continuous operation. This is a typical example of hybrid system. It is modelled in Simulink by using Sateflow diagrams to to handle the discrete changes of the control system, and linear differential equations to model the continuous behaviors.

The system contains four separate sensors: a throttle sensor, a speed sensor, an oxygen sensor, and a pressure sensor. Each of these sensors is represented by a parallel state in Stateflow, that is say finite state machines concurrently active. In total the entire logic of the systems is implemented by six parallel states. Each parallel state of a sensor contains two sub-states, a normal state and a fail state (the exception being the oxygen sensor, which also contains a warm-up state). If any of the sensor readings is outside an acceptable range, then a fault is registered, and the state of the sensor transitions to the failed sub-state. If the sensor recovers, it can transition back to the normal state.

In the original model, sensors faults are decided by the user using manual switch block for each sensor. The interest of the SMC approach comes from the possibility to observe a large set of execution traces produced by a probabilistic procedure. Therefor we replaced the Speed, EGO and MAP manual switches by custom probabilistic switches. These switches use a Poisson distribution and are parameterized by a rate to decide when a fault happen. A sensor will repair itself after a duration of 1 second. This modified model is similar to the one use in [45].

The Poisson distribution block that we use draws a random time T in seconds, that is the time before the next fault happens, and we use a Stateflow diagram as a timer. The signal from the Poisson block is then used by the sensor's switch. A Stateflow repair timer is used to maintain the fault signal for a duration of 1 second.

The system uses its sensors to maintain the air-fuel ratio at a constant value. When one sensor fails, a higher ratio is targeted to allow a smoother running. If another sensors fails the engine is shutdown for safety reasons, which is detected by a zero fuel rate.

We estimate the probability of a long engine shutdown. We use the following BLTL property to monitors executions over a period of 100 t.u., and to check if the fuel remains at zero for 1 t.u. :

$$\Phi = \neg F_{\leq 100}(G_{\leq 0.999}\mathsf{Fuel} = 0)$$

We try to reproduce with this property the results of [45]. In this paper they use a Bayesian SMC technique to estimate the probability of this property with the bound 1 for G operator. We can almost reproduce their results using the Monte Carlo algorithm on our own implementation of the Simulink model with stochastic distributions, but only if we use the approximated bound 0.999. Indeed the property is false, mainly when the three sensors are faulty at the same time. In that case the second sensor to fail remains in fault condition for exactly one second, with at least one other sensor. When this second sensor is repaired, there remains only one faulty sensor and the engine is restarted. Whether the Fuel variable in the sample after exactly one second is monitored at 0 or 1 by the SMC checker, changes the evaluation of the property. By using the value 0.999 we avoid these approximation issues. Table 1 recaps our results and the one of [45] for different values of the sensors fault rates (expressed in seconds). Our results are obtained with Plasma Monte Carlo (MC) algorithm after 1000 simulations. It takes approximately 2500 seconds to complete on a 2.7GHz Intel Core i7 with 8GB RAM and running MATLAB R2014b on Linux.

Fault rates	**Plasma MC**	**Bayesian SMC [45]**
(3 7 8)	0.396	0.356
(10 8 9)	0.748	0.853
(20 10 20)	0.93	0.984
(30 30 30)	0.985	0.996

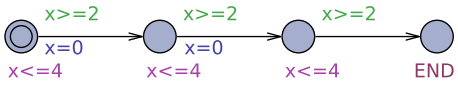
Table 1: Probability estimation of Φ with Plasma and the results from [45]. The fault rates in seconds correspond to the Speed, EGO and MAP sensors, respectively.

4 UPPAAL: Statistical Model Checking and Beyond

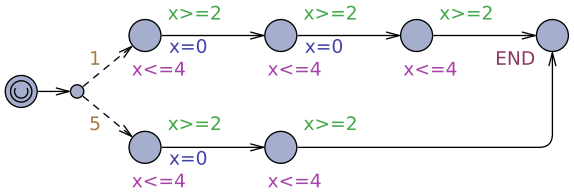
In the following we give an overview of the classical Uppaal toolbox (supporting model checking of timed automata base models), and the two recent branches Uppaal SMC and Uppaal Stratego, which supports statstical model checking of stochastic hybrid automata and synthesis learning for stochastic hybrid games, respectively.

4.1 UPPAAL

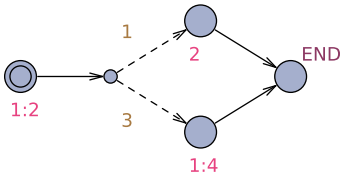
Uppaal is a toolbox for verification of real-time systems represented by (a network of) timed automata extended with integer variables, structured data types, and channel synchronization. The tool is jointly developed by Uppsala University



(a) A_1.



(b) A_2.



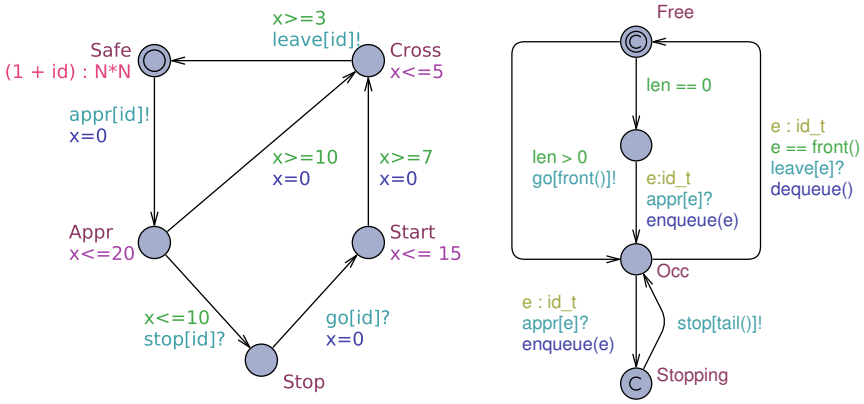
(c) A_3.

Fig. 4: Three stochastic timed automata.

and Aalborg University. It has been applied successfully in case studies ranging from communication protocols to multimedia applications (see [5] and [6] for concrete examples). The first version of UPPAAL was released in 1995 [34]. In the same spirit as any other professional model checker such as SPIN, UPPAAL proposes efficient data structures [36], a distributed version of UPPAAL [3,10], guided and minimal cost reachability [8, 9, 33], work on UML Statecharts [23], acceleration techniques [25], and new data structures and memory reductions [7, 11].

Example Consider the three TAs A_1, A_2 and A_3 from Fig. 4 each using a single clock x. Ignoring (initially) the weight annotations on locations and edges, the END-locations in the three automata are easily seen to be reachable within the time-intervals $[6, 12]$, $[4, 12]$ and $[0, +\infty)$.

Example To illustrate the extended input language of UPPAAL, we consider in Fig. 5 the Train Gate example adapted from [43]. The example model is distributed together with UPPAAL tool. A number of trains are approaching a bridge on which there is only one track. To avoid collisions, a controller stops the trains. It restarts them when possible to make sure that trains will eventually cross the bridge. There are timing constraints for stopping the trains modeling the fact that it is not possible to stop trains instantly. Each train has a designated clock x to constrain the timing between the different phases, e.g. the combination



(a) Train.

(b) Gate controller.

Fig. 5: Templates for the train-gate example.

of the invariant in the location Cross and the guard its ouToing edge models that once the passing of the crossing takes between 3 and 5 time units. Figure 5b shows the gate controller that keeps track of the trains with an internal queue data-structure (not shown here). It uses functions to queue trains (when a train approaches and the bridge is occupied in Occ) or dequeue them (when some train leaves and the bridge is free).

Queries The query language of UPPAAL consists of a subset of TCTL [1] allowing for reachability, safety and (time-bounded) liveness properties to be expressed.

Reachability properties are of the form `E<>`ϕ and means that there exists some path on which ϕ holds at some state. Reachability properties are useful for checking that models proposed at early design stages possess expected basic behaviours and to ask for diagnostic traces to confirm and study this more closely. For the Train Gate example such sanity properties could be:

```
E<> Gate.Occ
E<> Train(0).Cross
E<> Train(1).Cross
E<> Train(0).Cross and Train(1).Stop
E<> Train(0).Cross and (forall(i:id_t) i!=0 imply Train(i).Stop)
```

Safety properties are of the form `A[]`ϕ and mean that for all paths and for all states on those paths the property ϕ For the Train Gate example expected safety properties are:

```
A[] forall (i:id_t) forall (j:id_t) \
     Train(i).Cross && Train(j).Cross imply i==j
A[] not deadlock
```

Here the first safety property expresses that the gate controller correctly implements mutual exclusion of the bridge, in that no two different trains can be in the crossing simultaneously. The nested usage of the forall construct ranging over id t, ensures that the formula correctly (and conviniently) expresses mutual exclusion regardless of the number of trains.

Whereas safety properties are usefull for expressing "that something bad will never happen", they are not sufficient for ensuring that a designed system is adequate. Given the Train Gate example it is utterly simple to obtain a safe system guaranteeing no crashes on the bridge: simply use a gate controller that will stop all trains! Clearly, this is not satisfactory. What is needed is the additional ability to express liveness properties of a system in the sense "that something good is guaranteed to eventually happen". The first liveness property has the form `A<>`ϕ expressing that for all paths ϕ eventually holds. The second, and particularly useful, liveness property has the form ϕ`-->`ψ and should be read as ϕ leads to ψ in the sense that on any path starting in a reachable state where ϕ holds ψ will eventually hold. In our Train Gate example, we may want to ensure that whenever a train is approaching it eventually will be at crossing. This may be expressed by the following list of laads-to queries:

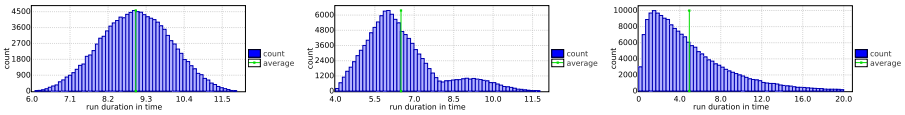
```
Train(0).Appr --> Train(0).Cross
Train(1).Appr --> Train(1).Cross
Train(2).Appr --> Train(2).Cross
...
```

4.2 UPPAAL SMC

Unfortunately, timed automata is not a panacea. In fact, albeit powerful, the model is not expressive enough to capture behaviors of complex cyber-physical systems. Indeed, the continuous time behaviors of those systems often rely on rich and complex dynamics as well as on stochastic behaviors. The model checking problem for such (stochastic hybrid) systems is in general undecidable, and approximating those behaviors with timed automata [26] was originally the best one could originally do in Uppaal.

With Uppaal SMC [22] we proposed an alternative to the above-mentioned problem. This new branch of Uppaal proposes to represent systems via networks of automata whose behaviors may depend on both stochastic and non-linear dynamical features. Concretely, in Uppaal SMC, each component of the system is described with an automaton whose clocks can evolve with various rates. Such rates can be specified with, e.g., ordinary differential equations. Moreover, each component chooses independently the point in time when it want to do its next discrete action, leading to a resulting *fully stochastic* combined system, with repeated time-races between the components.

To allow for the efficient analysis of probabilistic performance properties, Uppaal SMC proposes to work with Statistical Model Checking (SMC) [38, 44], an approach that has been proposed as an alternative to avoid an exhaustive exploration of the state-space of the model. The core idea of SMC is to monitor



(a) A_1 arrival to END. (b) A_2 arrival to END. (c) A_3 arrival to END.

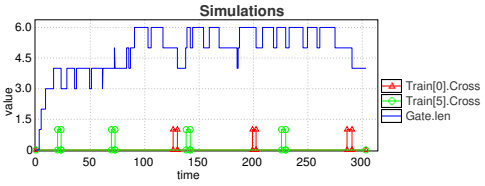
Fig. 6: Distributions of reachability time

random simulations of the system (obtained from its stochastic semantics), and then use classical the statistical methods (e.g. sequential hypothesis testing or Monte Carlo simulation) to decide whether the system satisfies the property with some desired degree of confidence.

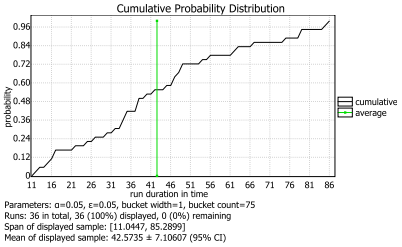
Modeling The modeling formalism of Uppaal SMC is based on a stochastic interpretation and extension of the timed automata (TA) formalism [2] used in the classical model checking version of Uppaal [5]. For *individual TA components* the stochastic interpretation replaces the non-deterministic choices between multiple enabled transitions by probabilistic choices (that may or may not be user-defined). Similarly, the non-deterministic choices of time-delays are refined by probability distributions, which at the component level are given either uniform distributions in cases with time-bounded delays or exponential distributions (with user-defined rates) in cases of unbounded delays.

Example Reconsider the three TAs A_1, A_2 and A_3 from Fig. 4. The stochastic interpretation of the three TAs provides probability distributions over the reachability time. For A_1, the delay of the three transitions will all be (automatically) resolved by independent, uniform distributions over $[2, 4]$. Thus the overall reachability time is given as the sum of three uniform distributions as illustrated in Fig. 6a. For A_2, the delay distributions determined by the upper and lower path to the END-location are similarly given by sums of uniform distributions. Subsequently, the combination ($\frac{1}{6}$ to $\frac{5}{6}$) of these as illustrated in distribution of the overall delay is obtained by a weighted Fig. 6b. Finally, in A_3 – in the absence of invariants – delays are chosen according to exponential distributions with user-supplied rates (here $\frac{1}{2}$, 2 and $\frac{1}{4}$). In addition, after the initial delay a discrete probabilistic choice ($\frac{1}{4}$ versus $\frac{3}{4}$) is made. The resulting distribution of the overall reachability time is given in Fig. 6c.

Importantly, the distributions provided by the stochastic semantics are in agreement with the delay intervals determined by the standard semantics of the underlying timed automata. Thus, the distributions for A_1 and A_2 have finite support by the intervals $[6, 12]$ and $[4, 12]$, respectively. Moreover, as indicated by A_3, the notion of stochastic timed automata encompasses both discrete and continuous time Markov chains. In particular, the class of distributions over reachability-time from the stochastic timed automata (STA) of Uppaal SMC includes that of phase-type distributions.



(a) Visualizing the gate length and when Train(0) and Train(5) cross on one random run.



(b) The cumulative probability distribution of `Pr[<=T](<> Train(5).Cross)`.

Fig. 7: Simulation and Distribution

Example Now reconsider the Train Gate example from Fig. 5. the interesing point w.r.t. SMC is the stochastic distribution of delays of the various trains in a given location. Figure 5a shows the template for a train. The location Safe has no invariant and defines the rate of the exponential distribution for delays. Trains delay according to this distribution and then approach by synchronizing on appr[i] with the gate controller. Here we define the rational $\frac{1+id}{N^2}$ where id is the identifier of the train and N is the number of trains. Rates are given by expressions that can depend on the current states. Trains with higher id arrive faster. Taking transitions from locations with invariants is given by a uniform distribution over the time interval defined by the invariant. This happens in locations Appr, Cross, and Start, e.g., it takes some time picked uniformly between 3 and 5 time units to cross the bridge.

Queries In addition to the standard model checking queries UPPAAL SMC provides a number of new queries related to the stochastic interpretation of timed automata. In particular UPPAAL SMC allows the user to visualize the values of expressions (evaluating to integers or clocks) along simulated runs, providinginsight to the user on the behavior of the system so that more interesting properties can be asked to the model-checker. On the Train Gate example we may monitor when Train(0) and Train(5) are crossing as well as the length of the queue. The query is

```
simulate 1 [<=300]
   { Train(0).Cross, Train(5).Cross, Gate.len }
```

This gives us the plot of Fig. 7a. Interestingly Train(5) crosses more often (since it has a higher arrival rate). Secondly, it seems unlikely that the gate length drops below 3 after some time (say 20), which is not an obvious property from the model.

For networks of stochastic timed automata the set of runs satisfying a property expressed in the linear-temporal logic MITL have a well-defined probability.

For (cost- or time-) bounded reachability properties and for bounded MITL properties these probabilities may be estimated using Monte Carlo simulation. The degree of confidence as well as the size of the confidence interval may be user-specified. Also, exploting sequential testing methods, such unknown property-probabilities may be tested against each other or against a given treshold. For the Train Gate example the following queries estimates that the Train(0) and Train(5) will be in the crossing before 100 time-units:

```
Pr[<=100](<> Train(0).Cross)
Pr[<=100](<> Train(5).Cross)
```

In fact with only 383 respectively 36 runs UPPAAL SMC returns the two 95% confidence intervals $[0.502421, 0.602316]$ and $[0.902606, 1]$. In addition more detailed information in terms of (cumulative, confidence interval, frequency histogram) probability distribution of the time-bounded reachability property, e.g. Fig. 7b.

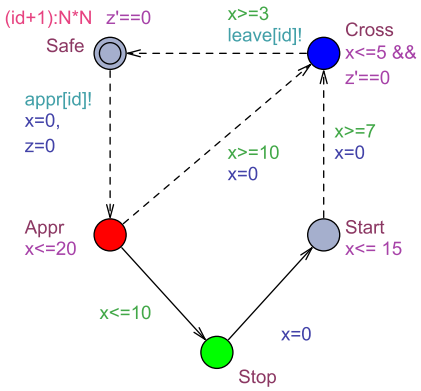
UPPAAL SMC has been applied to a wide range of case studies, going from systems biology [20, 21] to nash equilibrium analysis [13] or energy-centric systems [19, 42].

4.3 UPPAAL Stratego

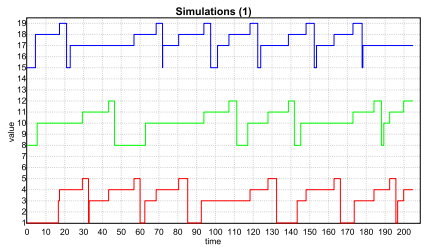
UPPAAL STRATEGO [16, 18] is a novel branch which facilitates generation, optimization, comparison as well as consequence and performance exploration of strategies for stochastic (priced) timed games in a user-friendly manner. In particular, UPPAAL STRATEGO (statistical model checking), UPPAAL TIGA [4] (synthesis for timed games) and the method proposed in [17] (synthesis of near optimal schedulers) have been integrated into one tool suite. UPPAAL STRATEGO comes with an extended query language where strategies are first class objects that may be constructed, compared, optimized and used when performing (statistical) model checking of a game under the constraints of a given synthesized strategy. Thus, the tool allows for efficient and flexible "strategy-space" exploration before adaptation in a final implementation by maintaining strategies as first class objects in the model-checking query language.

Example Now consider a *game* version of the Train Gate example given in Fig. 8a, between an environment consisting of the various trains – with uncontrollable behaviour in terms of when to approach, and choice of time for crossing indicated by dashed transitions – and the control options for the gate – with controllability of stopping and restarting of trains indicated by full transitions. The aim is to synthesize a control strategy for the gate given a specified objective.

Assuming that the trains (i.e. the environment) chooses their delays according to the specified distributions (uniform or exponential) the game is really a $\frac{1}{2}$-player game (i.e. and infinite-state MDP), where the objective of the controller would be to optimize some cost-function. However, as illustrated in Fig. 9 we can abstract the $\frac{1}{2}$-player game into a 2-player timed game simply by ignoring



(a) Train Game.



(b) Simulation run for `Safe`

Fig. 8: Train Game

the stochasticity and the (possible) price-decoration. In particular, given a safety (including time-bound reachability) objective ϕ, we may use the branch Uppaal Tiga to synthesize a most permissive, non-deterministic and memory-less strategy σ which ensure that the objective ϕ is met. Constraining the game $\mathcal{G}$ with respect to σ it is possible to perform additional (statistical) model checking of under the strategy using Uppaal and Uppaal SMC. For the Train Game example an obvious safety strategy is that no two different trains should ever be in the crossing at the same time. The following Uppaal Stratego query asks for a strategy ensuring this objective to be synthesized:

```
strategy Safe =
    control: A[] forall (i : id_t) forall (j : id_t)
                 Train(i).Cross && Train(j).Cross imply i == j
```

Uppaal Stratego answers affirmative to this query and returns a non-deterministic strategy (named `Safe`) that ensures the objective. Now given a synthesized strategy (essential) all Uppaal and Uppaal SMC queries may be performed on the game restricted with the strategy. In particular – having added

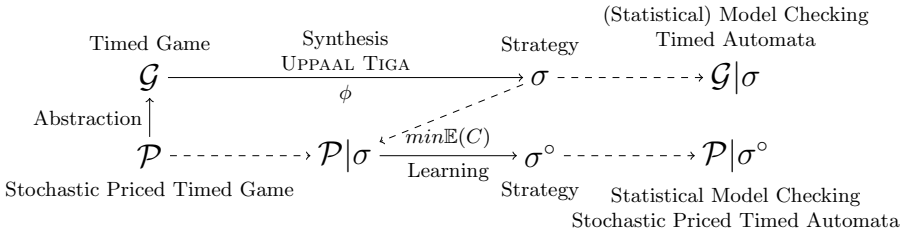


Fig. 9: Overview of models and their relations. The lines show different actions. The dashed lines show that we use the object.

a local clock z – we may estimate the expected time from any particular train enters location Appr until it reaches Cross. The query:

```
E[<=200 ; 100] (max: Train(0).z) under Safe
```

estimates this expected time for Train(0) to 40.7866 ± 2.58574, and the following query:

```
simulate 1 [<=200]
  {  Train(0).Safe + 2*Train(0).Appr + 3*Train(0).Stop
                + 4*Train(0).Start + 5*Train(0).Cross,
     7+Train(1).Safe + 2*Train(1).Appr + 3*Train(1).Stop
                + 4*Train(1).Start + 5*Train(1).Cross,
     14+Train(2).Safe + 2*Train(2).Appr + 3*Train(2).Stop
                + 4*Train(2).Start + 5*Train(2).Cross
  } under Safe
```

generates a random run of duration 200 tracing three expressions that each indicate numerically the state of a one of three trains as illustrated in the plot of Fig. 8b. As can be seen (noticing that for each train the max value indicates that the particular train is in the crossing) – and as expected given the objective – there are never two different trains in the crossing at the same time. One objection to this strategy is that the expected time for Train(0) to enter the crossing is to high. To remedy this on may apply reinforcement learning as implemented in Uppaal Stratego for the stochastic priced timed game to obtain a strategy which minimizes this expectation. The following query:

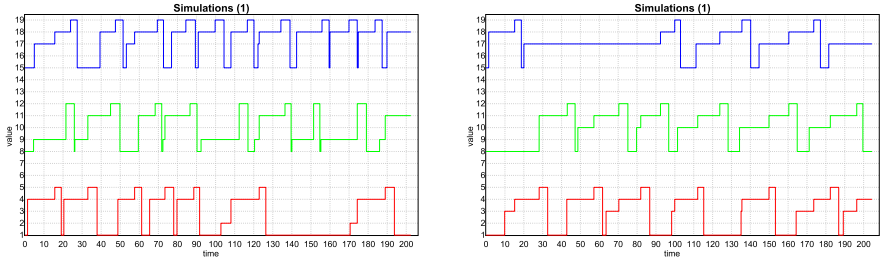
```
strategy GoFast =
      minE (Train(0).z) [<=100] : <> Train(0).Cross
```

leads in a total of 88 iterations to a strategy `GoFast` for which the expected time is only 15.6394 ± 0.411427 (thus a substantial improvement). However as can be seen from the plot the this strategy – though near-ideal in performance – is in no way safe, as there are several instances of this single random run where two different trains are simultaneously in the crossing.

Fortunately reinforcement learning strategies may also be subject to the restriction of already generated safety strategies. Thus the query:

```
strategy GoFastSafe =
      minE (Train(0).z) [<=100] : <> Train(0).Cross
             under Safe
```

uses reinforcement learning with a total of 42 iterations to generate the substrategy `GoFastSafe`. Here the expected time for Train(0) to reach the crossing is again attempted minimized, but now within the boundary of what are permitted by the safety strategy `Safe`. Now the expected time is 22.1332 ± 0.494325, which is still a lot better than the `Safe` strategy, but with the guarantee that safety is met in contrast to the `GoFast` strategy. The plot in Fig. 10b witnesses this. As can be seen from this plot, the optimization for Train(0) is on the expense of the performance for the other trains.



(a) Simulation run for `GoFast` (b) Simulation run for `GoFastSafe`

Fig. 10: Simulation Runs

References

1. R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Inf. Comput.*, 1993.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 1994.
3. G. Behrmann. Distributed reachability analysis in timed automata. *STTT*, 2005.
4. G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-Tiga: Time for playing games! In *Computer Aided Verification*, vol. 4590 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007.
5. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *QEST*. IEEE Computer Society, 2006.
6. G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Developing uppaal over 15 years. *Softw., Pract. Exper.*, 2011.
7. G. Behrmann, A. David, K. G. Larsen, and W. Yi. Unification & sharing in timed automata verification. In *SPIN*, vol. 2648 of *LNCS*, 2003.
8. G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient guiding towards cost-optimality in UPPAAL. In *TACAS*, number 2031 in LNCS. Springer, 2001.
9. G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced timed automata. In *HSCC*, number 2034 in LNCS. Springer, 2001.
10. G. Behrmann, T. Hune, and F. Vaandrager. Distributed timed model checking - How the search order matters. In *CAV*, LNCS, Chicago, 2000. Springer-Verlag.
11. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference d iagrams. In *CAV*, vol. 1633 of *LNCS*. Springer–Verlag, 1999.
12. B. Boyer, K. Corre, A. Legay, and S. Sedwards. PLASMA-lab: A flexible, distributable statistical model checking library. In *QEST*, vol. 8054 of *LNCS*. Springer Berlin Heidelberg, 2013.
13. P. E. Bulychev, A. David, K. G. Larsen, A. Legay, and M. Mikucionis. Computing nash equilibrium in wireless ad hoc networks: A simulation-based approach. In *IWIGP*, vol. 78 of *EPTCS*, 2012.
14. A. Colombo, D. Fontanelli, D. Gandhi, A. De Angeli, L. Palopoli, S. Sedwards, and A. Legay. Behavioural templates improve robot motion planning with social force model in human environments. In *EFTA*. IEEE, 2013.

15. A. Colombo, D. Fontanelli, A. Legay, L. Palopoli, and S. Sedwards. Motion planning in crowds using statistical model checking to enhance the social force model. In *CDC*. IEEE, 2013.
16. A. David, P. G. Jensen, K. G. Larsen, A. Legay, D. Lime, M. G. Sørensen, and J. H. Taankvist. On time with minimal expected cost! In *ATVA*, vol. 8837 of *LNCS*. Springer, 2014.
17. A. David, P. G. Jensen, K. G. Larsen, A. Legay, D. Lime, M. G. Sørensen, and J. H. Taankvist. On time with minimal expected cost! In *ATVA*, 2014.
18. A. David, P. G. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist. Uppaal stratego. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, vol. 9035 of *Lecture Notes in Computer Science*. Springer, 2015.
19. A. David, K. G. Larsen, A. Legay, and M. Mikucionis. Schedulability of herschel revisited using statistical model checking. *STTT*, 2015.
20. A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, and S. Sedwards. Runtime verification of biological systems. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, LNCS, 2012.
21. A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, and S. Sedwards. Statistical model checking for biological systems. *STTT*, 2015.
22. A. David, K. G. Larsen, A. Legay, M. Mikucionis, and Z. Wang. Time for statistical model checking of real-time systems. In *CAV*, vol. 6806 of *LNCS*. Springer, 2011.
23. A. David, M. O. Möller, and W. Yi. Formal verification of UML statecharts with real-time extensions. In *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002*, vol. 2306 of *LNCS*. Springer–Verlag, 2002.
24. D. Helbing and P. Molnár. Social force model for pedestrian dynamics. *Phys. Rev. E*, 1995.
25. M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. In *Electronic Notes in Theoretical Computer Science*, vol. 65. Elsevier Science Publishers, April 2002.
26. T. A. Henzinger and P. Ho. Algorithmic analysis of nonlinear hybrid systems. In *CAV*.
27. T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *VMCAI*, vol. 2937 of *LNCS*. Springer, 2004.
28. C. Jegourel, A. Legay, and S. Sedwards. A Platform for High Performance Statistical Model Checking – PLASMA. In *TACAS*, vol. 7214 of *LNCS*. Springer, 2012.
29. C. Jegourel, A. Legay, and S. Sedwards. Cross-Entropy Optimisation of Importance Sampling Parameters for Statistical Model Checking. In *CAV*, vol. 7358 of *LNCS*. Springer, 2012.
30. C. Jegourel, A. Legay, and S. Sedwards. Importance splitting for statistical model checking rare properties. In *Computer Aided Verification*. Springer, 2013.
31. C. Jégourel, A. Legay, S. Sedwards, and L. Traonouez. Distributed verification of rare properties using importance splitting observers. *ECEASST*, 2015.
32. H. Kahn and A. W. Marshall. Methods of Reducing Sample Size in Monte Carlo Computations. *Operations Research*, November 1953.

33. K. G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *CAV*, number 2102 in LNCS. Springer, 2001.
34. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, Oct. 1997.
35. K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inf. Comput.*, 1991.
36. F. Larsson, K. G. Larsen, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, Dec. 1997.
37. M. Okamoto. Some inequalities relating to the partial sum of binomial probabilities. *Annals of the Institute of Statistical Mathematics*, 1959.
38. K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, LNCS 3114. Springer, 2004.
39. K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *CAV*, LNCS 3576, 2005.
40. K. Sen, M. Viswanathan, and G. A. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *QEST*. IEEE Computer Society, 2005.
41. A. Wald. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 1945.
42. E. R. Wognsen, B. R. Haverkort, M. R. Jongerden, R. R. Hansen, and K. G. Larsen. A score function for optimizing the cycle-life of battery-powered embedded systems. In *Formal Modeling and Analysis of Timed Systems - 13th International Conference, FORMATS 2015, Madrid, Spain, September 2-4, 2015, Proceedings*, vol. 9268 of *LNCS*. Springer, 2015.
43. W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, London, UK, UK, 1995. Chapman & Hall, Ltd.
44. H. L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon, 2005.
45. P. Zuliani, A. Platzer, and E. M. Clarke. Bayesian statistical model checking with application to Stateflow/Simulink verification. *Formal Methods in System Design*, 2013.